Debugging Standard Document Formats

Nabil Layaïda, Pierre Genevès

Debugging Standard Document Formats

Nabil Layaïda
INRIA
France
nabil.layaida@inria.fr

Pierre Genevès
CNRS
France
pierre.geneves@inria.fr

ABSTRACT

We present a tool for helping XML schema designers to obtain a high quality level for their specifications. The tool allows one to analyze relations between classes of XML documents and formally prove them. For instance, the tool can be used to check forward and backward compatibilities of recommendations. When such a relation does not hold, the tool allows one to identify the reasons and reports detailed counter-examples that exemplify the problem. For this purpose, the tool relies on recent advances in logic-based automated theorem proving techniques that allow for efficient reasoning on very large sets of XML documents. We believe this tool can be of great value for standardization bodies that define specifications using various XML type definition languages (such as W3C specifications), and are concerned with quality assurance for their normative recommendations.

Categories and Subject Descriptors

I.7.2 [**Document and Text Processing**]: Document Preparation—*markup languages*; D.3.2 [**Programming Languages**]: Language Classifications—*extensible languages*

General Terms

Algorithms, languages, theory, verification

Keywords

XML, schema, format, compatibility.

1. INTRODUCTION

In the document world, a driving concern is the long term access to content. That is the need to be able to process content, for example a web page, a scalable vector graphic, written today, in say several decades. This major concern gave birth to SGML, and, more recently to XML, where the idea is to separate data structures from processor-specific instructions. For this purpose, the essence of XML consists in organizing information in tree-tagged structures conforming to some constraints, which are expressed using standard type definition languages such as DTDs, XML schemas and Relax NG. A set of constraints define a class of documents.

One major role of organizations such as W3C is to contribute to the standardization effort leading to a unique widely accepted set of constraints for a given class of documents. Designing a normative specification is a complex process, which is made even harder by a few important considerations.

First, when a language is designed, one need to take into account how future versions of that language can evolve. For a particular version of a language, not only the schema constraints allowed by that version need to be considered but also how they can be modified by future versions. This allows to address how an implementation of this version should process document variants added by future schema versions.

Second, specification authors must provide a formal description of their recommendation as a schema. Such schemas may be written in languages such as DTD, XML Schema, or Relax-NG, to name only the most popular ones. It is common practice to describe the same structure, or new versions of a structure, in different schema languages. Document formats developed by W3C provide a variety of examples: XHTML 1.0 has both DTDs and XML Schemas, while XHTML 2.0 has a Relax-NG definition; the schema for SVG Tiny 1.1 is a DTD, while version 1.2 is written in Relax-NG; MathML 1.01 has a DTD, MathML 2.0 has both a DTD and an XML Schema, and MathML 3.0 is developed with a Relax-NG and is expected to have also a DTD and an XML Schema.

An issue is then to make sure that schemas written in different languages are equivalent, *i.e.* they describe the same structure, possibly with some differences due to the expressivity of the language for defining tree grammar based constraints [7, 3, 2]. Otherwise, another issue is to precisely identify the differences between two versions of the same schema expressed in different languages. Moreover, the issues raised by forward and backward compatibility of document instances obviously remain when schema languages change from a version to another.

Specifically, we identify three different properties for a normative specification:

- *Forward compatibility*: All instances of an older specification should be valid with respect to newer specifications. This ensures that a document can still be processed properly with applications implementing newer specifications.
- *Backward compatibility without added elements/attributes*: New combinations of old elements are not supposed to be introduced in later specifications. Otherwise, an application implementing an older specification will not able to process a document that conforms to some future specification, even if this document does not contain any element or attribute introduced as extensions.
- *Equivalence between schema versions*: We expect the different schema versions of the same recommendation to define the same set of documents modulo the expressivity of the schema language.

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

An XML schema definition (whether normative or not) often evolves over time, as new needs often result in new features usually introduced as new elements and attributes. However we believe that this normal evolution should not break the three previous properties. In this paper, we present a tool capable of checking each of these properties expressed through a predicate language introduced in the next section.

2. THE PREDICATE LANGUAGE

The demo presented in this paper is a tool which uses predicates offered to XML application and recommendation designers for assessing the correctness of schema evolutions, with respect to the properties described above. These predicates can be combined with logical connectives (e.g. disjunction, conjunction, negation...) in order to give the focus on particular peculiarities of a schema and its variants.The demo scenario consists in specifying and testing the satisfiability of theses predicates applied to W3C Document Recommendations (XHTML, SVG, MathML and SMIL). An unsatisfiable predicate means that the property expressed by that predicate holds, otherwise the system generates automatically a counter-example document which proves that the property does not hold.

The syntax of the predicate language that can be used for XML reasoning is shown in Figure 1, where a *QName* denotes any XML qualified name. The predicate `type`("*file*", *QName*) allows one to refer to an existing specification defined in a variety of type definition languages including DTD, XML Schema, and Relax NG. The first argument of this predicate, *file*, is a file path to the schema file and the second argument, *QName*, is the element name to be considered as the entry point (root) of the given schema.

The predicate `added_element`(*file*, *file*′, *QName*) takes three parameters: *file* and *file*′ are file paths to schema specifications, and *QName* is an XML element name to be considered as the document root. This predicate corresponds to the set of all element names defined in *file*′ but not in *file* (or in other terms, elements that were added in *file*′ compared to *file*). In a similar manner, the predicate `added_attribute`(*file*,*file*',*QName*) defines the set of all attribute names introduced in *file*′ compared to *file*.

The predicate `backward_incompatible`(*file*, *file*′, *QName*) takes two type expressions as parameters, where *file*′ is a new version of *file*. This predicate is unsatisfiable iff all instances of *file*′ are also valid against *file*.

The predicate `exclude`(*QName*) can be used for excluding a specific element name. This predicate can also be used for checking properties in an iterative manner, refining the property to be tested at each step. For example, one can check the backward compatibility without added elements with the following formula that combines two predicates using a simple conjunction:

`backward_incompatible`(*file*, *file*′, *QName*) &
`exclude`(*QName*′)

3. DEMONSTRATION OVERVIEW

The system has been implemented as a Java/JSP web application and interaction with the system is offered through a web user interface in a web browser. The tool is available online from:

`http://wam.inrialpes.fr/xml`

Our demonstration aims to showcase schema analyses progressively by refining a set of predicates (by combining them with boolean operators) across a variety of use cases. The user can either enter an analysis problem using predicates through area (1) of Figure 2 or select from pre-loaded analysis tasks offered in area (4) of Figure 2. The level of details displayed by the analyzer can be adjusted in area (2) of Figure 2 and allows to inspect logical translations and statistics on problem size and the different operation costs. The results of the analysis are displayed in area (3) of Figure 2 together with XML counter-examples.

formula ::=

`type`("*file*", *QName*)	
`added_element`(*file*, *file*′, *QName*)	
`added_attribute`(*file*, *file*′, *QName*)	
`forward_incompatible`(*file*, *file*′, *QName*)	
`backward_incompatible`(*file*, *file*′, *QName*)	
`descendant`(*QName*)	
`exclude`(*QName*)	
formula | *formula*	(disjunction)
formula & *formula*	(conjunction)
~*formula*	(negation)
formula => *formula*	(implication)
formula <=> *formula*	(equivalence)
QName	

Figure 1: High-Level Language for XML Reasoning.

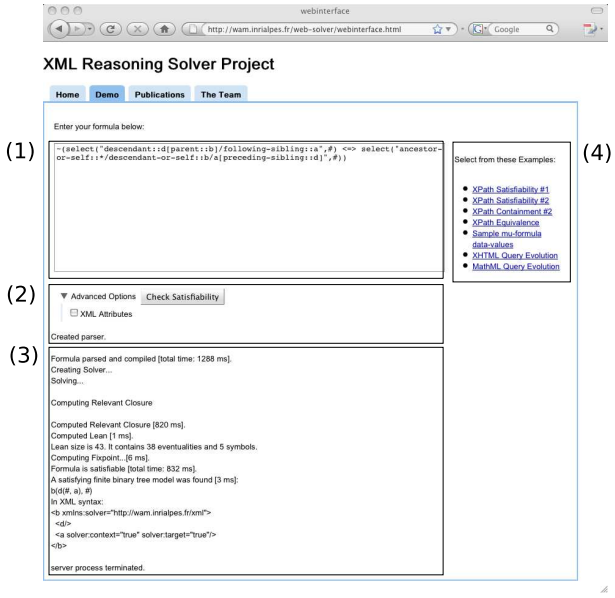

Figure 2: Screenshot of the Solver Interface.

Predicates are in fact translated automatically by our tool to logical formulas used by a solver we developed in an earlier work (see [5, 1] for the underlying logic and its satisfiability-testing algorithm). As a result, the low level logical formulation and resolution remain transparent to the XML schema designer. Additional predicates related to the analysis of XPath expressions can be found in [4]. During the demonstration, we show that the tool works well in terms of performance on the fairly large schemas such as XHTML, MathML, and SMIL. Once a counter-example is generated, we provide it to an external validator [6] together with W3C document

schemas to prove that it corresponds to a genuine bug.

XHTML Basic

The first test consists in analyzing the relationship (forward and backward compatibility) between XHTML basic 1.0 and XHTML basic 1.1 schemas. In particular, backward compatibility can be checked by the following command:

```
backward_incompatible("xhtml-basic10.dtd",
                      "xhtml-basic11.dtd", "html")
```

Executing the test yields a counter example as the new schema contains new element names. The counter example (shown below) contains a `style` element occurring as a child of `head`, which is not permitted in XHTML basic 1.0:

```
<html>
  <head>
    <title/>
    <style type="_otherV"/>
  </head>
  <body/>
</html>
```

The next step consists in focusing on the relationship between both schemas excluding these new elements. This can be formulated by the following command:

```
backward_incompatible("xhtml-basic10.dtd",
                      "xhtml-basic11.dtd", "html")
& exclude(added_element(
      type("xhtml-basic10.dtd","html"),
      type("xhtml-basic11.dtd", "html")))
```

The result of the test shows a counter example document that proves that XHTML basic 1.1 is not backward compatible with XHTML basic 1.0 even if new elements are not considered. In particular, the content model of the `label` element cannot have an `a` element in XHTML basic 1.0 while it can in XHTML basic 1.1. The counter example produced by the solver is shown below:

```
<html>
  <head>
    <object>
      <label>
        <a>
          <img/>
        </a>
        <img/>
      </label>
      <param/>
    </object>
    <meta/>
    <title/>
    <base/>
  </head>
  <body/>
</html>

XTML basic 1.0 validity error: element a is not
declared in label list of possible children
```

SMIL

The second test consists in analyzing the relationship (forward and backward compatibility) between several versions of the SMIL standard[1], namely versions 1.0, 2.0, and 3.0. In particular, forward compatibility between 1.0 and 2.0 can be checked by the following command:

[1]The first author was a member of the W3C SMIL working group and co-author of SMIL 2.0 and 2.1.

```
forward_incompatible("SMIL10.dtd", "SMIL20.dtd", "smil")
```

The result of the test shows a counter example document that proves that there exist valid SMIL 1.0 documents that are not valid anymore with respect to SMIL 2.0. In fact that is because the content model of the `layout` element is defined as `any` in SMIL 1.0, whereas it is more restricted in SMIL 2.0. We observe that introducing `any` is a choice that has important consequences. Indeed, a document that was playable with 1.0 implementations may no longer be playable using 2.0 implementations. The counter example produced by the solver is shown below:

```
<smil>
  <head>
    <layout>
      <meta content="_otherV" name="_otherV"/>
    </layout>
  </head>
</smil>

SMIL 2.0 validity error:
Element layout content does not follow the DTD,
expecting (region|topLayout|root-layout|regPoint)*,
got (meta)
```

The lesson here is that introducing very permissive content models (like `any`) has to be considered very seriously. Indeed, that means that all future version of the standard should be at least as permissive. Otherwise, all content produced with earlier (more permissive) versions becomes at risk. Therefore, the initial content model has to be carefully designed in order to avoid such situations.

The following example is even worse. We check forward compatibility between SMIL 2.0 and 3.0:

```
forward_incompatible("SMIL20.dtd",
                          "SMIL30Language.dtd", "smil")
```

We obtain the following counter-example:

```
<smil xmlns="http://www.w3.org/2001/SMIL20/Language">
  <body>
    <switch>
      <animateMotion/>
    </switch>
    <a href="..."/>
  </body>
</smil>
```

This document is valid with respect to SMIL 2.0. However it does not validate with respect to SMIL 3.0. That is because the content model for the `switch` element was set to a more restrictive pattern in version 3.0 compared to 2.0, as the validation error message suggests below:

```
SMIL 3.0 validity error :
Element switch content does not follow the DTD,
expecting ((metadata | switch)* , ((((animate | set |
animateMotion | animateColor) , (metadata | switch)*)* ,
(((par | seq | excl | audio | video | animation | text |
 ... switch)*)+)) | (layout , (metadata | switch)*)*)),
got (animateMotion)
```

We would like to know if the bug is limited to the occurrence of the `animateMotion` element or whether it is more general. To this end, we progressively exclude elements named `animateMotion`, `set`, `animateColor`, and `animate`, as follows:

```
forward_incompatible("SMIL20.dtd",
                          "SMIL30Language.dtd", "smil")
  & exclude(animateMotion) & exclude(set)
  & exclude(animateColor) & exclude(animate)
```

We still obtain the following counter-example (valid w.r.t SMIL 2.0 but not w.r.t SMIL 3.0), which shows that the forward incompatibility is not limited to the occurence of the previous elements, but rather, to severe limitations of the `switch` content model introduced in 3.0. In other words, `switch` is an element which undermines SMIL forward compatibility.

```
<smil xmlns="http://www.w3.org/2001/SMIL20/Language">
  <body>
    <switch>
      <seq/>
      <area/>
    </switch>
    <switch/>
    <a href="..."/>
  </body>
</smil>
```

SVG

The SVG test consists in analyzing the relationship (forward and backward compatibility) between SVG 1.0 et 1.1. In particular, we examine the differents profiles (tiny, basic and full) from 1.0 and compared to 1.1 schemas. Backward compatibility can be checked by the following command:

```
forward_incompatible("svg10.dtd",
             "svg11-flat-20030114.dtd", "svg")
```

The test is unsatisfiable meaning that SVG 1.1 is formally proven to be forward compatible with SVG 1.0. This is good news as it means that all 1.0 documents will be supported with 1.1 conforming implementations, without any exception. In the case where a 1.0 document does not play with a 1.1 implementation, this indicates a bug in the implementation and not in the SVG specification.

We observe here that the common practice of including a single doctype declaration within a document is questionable, since a document is not only valid w.r.t a given schema but also w.r.t to all future forward-compatible versions. Keeping track of this mapping between a document and several schemas allows the document to be supported by a larger set of implementations.

Similar tests on the SVG 1.1 tiny, basic and full also exhibit good results. This corresponds to the definition of these three profiles as strict subsets of each other. Furthermore, we believe that the use of a modularized version of a schema (as opposed to a complete redefinition) has helped in avoiding compatibility problems.

We now focus on testing the backward compatibility between the SVG basic 1.1 profile and SVG 1.0 profile. The test fails even if new features are left aside:

```
backward_incompatible("svg10.dtd",
            "svg11-basic.dtd", "svg")
& exclude( added_element(type("svg10.dtd", "svg"),
                         type("svg11-basic.dtd","svg")))
& exclude(switch)
```

This test yields the following counter-example which confirms that there is actually a flaw in the 1.1 specification:

```
<svg>
  <image>
    <title/>
    <title/>
  </image>
</svg>
```

as it allows two `title` elements to occur inside an `image` element, which was not allowed in the 1.0.

MathML

We apply a similar investigation approach to MathML 1.0 and its newer version 2.0. We formulate a backward compatibility test without elements that were added in version 2.0. Furthermore, we want to exclude immediate trivial counter-examples involving the use of the `declare` element as well as of the `math` element occuring within the `anootation-xml` element. For this purpose, we use the following formulation:

```
backward_incompatible("mathml.dtd","mathml2.dtd","math")
& exclude( added_element( type("mathml.dtd","math"),
                          type("mathml2.dtd","math")))
& exclude(declare)
& (~descendant(math))
```

that bans the `declare` element from occuring in the whole tree (achieved with the use of the `exclude(declare)` predicate), and prevents the `math` element from ocurring in the root's subtree (owing to the use of the `(~descendant(math))` predicate) The following counter-example is produced:

```
<math>
  <apply>
    <annotation-xml>
      <mprescripts/>
    </annotation-xml>
  </apply>
</math>
```

Such backward incompatibilities suggest that applications cannot simply ignore new elements from newer schemas, as the combination of older elements may evolve significantly from one version to another.

4. CONCLUSION

In this demo paper, we illustrated how to use the tool on real-world XML schema specifications produced by W3C. We show that the tool can be very useful for designers of normative recommendations, in order to assist them for detecting bugs and enforcing some level of quality assurance. The tool also allows document authors and content providers to identify language features that are source of forward and backward compatibility concerns. The demo also shows the usefulness of new progress made in formal language theory applied to practical and critical issues such as compatibilities of standard formats. Such issues jeopardize the future access in the long term to billions of documents written today. If one cannot fix every broken or non-compliant implementation of standard recommendations, we now have tools to help the design of compatible versions of recommendations.

5. REFERENCES

[1] P. Genevès. *Logics for XML: Reasoning with Trees.* VDM Verlag, September 2009.

[2] P. Genevès and N. Layaïda. A system for the static analysis of XPath. *ACM TOIS*, 24(4):475–502, October 2006.

[3] P. Genevès and N. Layaida. Deciding XPath containment with MSO. *DKE*, 63(1):108–136, October 2007.

[4] P. Genevès, N. Layaïda, and V. Quint. Identifying query incompatibilities with evolving XML schemas. In *ICFP'09*.

[5] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *PLDI'07*.

[6] Libxml2. http://www.xmlsoft.org/.

[7] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM TOIT*, 5(4):660–704, 2005.